Centre de recherche sur les transports (C.R.T.)
Centre for Research on Transportation

CRT-2005-36

VARIABLE NEIGHBORHOOD SEARCH FOR THE PICKUP AND DELIVERY TRAVELING SALESMAN PROBLEM WITH LIFO LOADING

by

Francesco Carrabs
Jean-François Cordeau
Gilbert Laporte

December 2005

Université de Montréal
C.P. 6128, succursale Centre-ville, Montréal QC H3C 3J7 Canada
Téléphone: (514) 343-7575 / Télécopieur: (514) 343-7121

crt@crt.umontreal.ca
www.crt.umontreal.ca

HEC ÉCOLE POLYTECHNIQUE MONTRÉAL Université de Montréal

106-02188/

PUBLICATION
CRT-2005-36

VARIABLE NEIGHBORHOOD SEARCH FOR THE PICKUP AND DELIVERY TRAVELING SALESMAN PROBLEM WITH LIFO LOADING

by

Francesco Carrabs[1]
Jean-François Cordeau[2]
Gilbert Laporte[2]

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[1] Dipartimento di Matematica ed Informatica, Università di Salerno, 84084 Fisciano (SA), Italy (fcarrabs@unisa.it)

[2] Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7, and Canada Research Chair in Distribution Management, HEC Montréal, 3000, Côte-Sainte-Catherine, Montréal, Canada H3T 2A7 (cordeau, gilbert@crt.umontreal.ca)

Dépôt légal – Bibliothèque nationale du Québec
Bibliothèque nationale du Canada, 2005

Abstract

This paper addresses a variation of the traveling salesman problem with pickup and delivery in which loading and unloading operations have to be executed in a LIFO (Last-in-First-Out) order. We introduce three new local search operators for this problem which are then embedded within a variable neighborhood search heuristic. We evaluate the performance of the heuristic on data adapted from TSPLIB instances.

Keywords: traveling salesman problem, pickup and delivery, LIFO loading, variable neighborhood search.

Résumé

Cet article traite d'une variante du problème du voyageur de commerce avec cueillettes et livraisons dans lequel les opérations de chargement et de déchargement du véhicule doivent être faites dans l'ordre « dernier entré, premier sorti» (LIFO). Nous présentons trois nouveaux opérateurs de recherche locale qui sont ensuite utilisés à l'intérieur d'une heuristique de recherche à voisinage variable. Nous évaluons la performance de l'heuristique sur des données adaptées d'instances de TSPLIB.

Mots-clés: problème du voyageur de commerce, cueillettes et livraisons, LIFO, recherche à voisinage variable.

1. Introduction

This paper concerns a variation of the *Traveling Salesman Problem with Pickup and Delivery* (TSPPD) called the *TSPPD with LIFO Loading* (TSPPDL). The TSPPD is well known. It consist of determining a minimum length tour traveled by a vehicle to service n requests. Each request is characterized by an origin vertex, the *pickup* location, where goods are loaded, and a destination vertex, the *delivery* location, where goods are unloaded. The vehicle starts from a fixed vertex, the *depot*, and returns to it after all requests have been satisfied. Every other vertex has to be visited exactly once, with the additional constraint that the pickup vertex associated with any given request must be visited before the corresponding delivery vertex. This problem has been studied, among others, by Kalantari et al. (1985), Fischetti and Toth (1989), Savelsbergh (1990), Healy and Moll (1995), Ruland and Rodin (1997), Renaud, Boctor and Ouenniche (2000), Renaud, Boctor and Laporte (2002). For a recent survey see Cordeau et al. (2006).

In the TSPPDL, the LIFO (Last-in-First-Out) constraint states that the loading and unloading operations must be executed in a LIFO order. This means that when loading, the goods are always placed at the rear. Similarly, unloading at a delivery customer is only allowed if the goods of the current delivery are at the rear of the load. The TSPPDL can be formally stated as follows. Let $R = \{1, ..., n\}$ be a set of n requests. A request $x \in R$ is composed of a pickup vertex x^+ and a delivery vertex x^-. Let $P = \{1^+, ..., n^+\}$ be the set of pickup vertices and $D = \{1^-, ..., n^-\}$ the set of delivery vertices. We denote the depot by 0 or $2n + 1$. The TSPPDL is defined on a weighted complete digraph $G = (N, A, c)$, where $N = P \cup D \cup \{0\}$ is the vertex set, A is the arc set, and c is the cost function defined on A. The cost of arc (x, y) is denoted by $c(x, y)$. The problem is to determine a minimum cost Hamiltonian cycle (or *tour*) on G, subject to the constraint that the pickup and delivery operations are executed in a LIFO fashion.

The TSPPDL has several practical applications. Indeed, vehicles often have only one door at the rear, which means that a LIFO policy is more convenient, especially when delivering large objects like furniture. Avoiding unnecessary handling is also important when delivering hazardous materials. There is only a limited literature on the TSPPDL. Cassani and Righini (2004) have presented a variable neighborhood descent (VND) heuristic based on four improvement operators called *couple-exchange*, *block-exchange*, *relocate-couple* and *relocate-block*. These operators will be described in detail in the next section. Pacheco Bonrostro (1997) has adapted to the TSPPDL the Or-opt operator (Or, 1976) for the *Traveling Salesman Problem* (TSP). This operator relocates chains of one, two or three vertices in different positions in the tour. The total number of possible exchanges is $\theta(n^2)$, but the Pacheco Bonrostro adaptation runs in $\theta(n^3)$ time due to the checks needed to find feasible 3-exchanges for the TSPPDL. Van Der Bruggen, Lenstra and Schuur (1993) have introduced a variable depth search heuristic for the TSPPD with time windows, based on seven arc-exchange procedures. We will use some of these procedures in our heuristic. Ladany and Mehrez (1984) have studied a version of the TSPPDL in which the LIFO constraint is relaxed, and its violations are penalized in the objective function. Computational results are presented for very small instances only. Xu et al. (2003) have studied a multiple-vehicle pickup and delivery problem with a LIFO constraint, multiple time windows, compatibility constraints, and a complex cost structure. The authors have proposed a column generation based heuristic for this problem. Volchenkov (1982) has analyzed a planar layout problem with LIFO constraints. The results were later used by Levitin (1986) and Levitin and Abez-

gaouzb (2003). The latter paper proposes an exact algorithm for the routing of multiple-load automated guided vehicles. This problem is in fact a TSPPDL with the difference that each pickup customer can be associated with more than one delivery customer, and vice-versa.

In this paper, we present three new operators: *multi-relocate*, *2-opt-L*, and *double-bridge*. The first one is entirely new, while the last two are adaptions of the classical 2-opt and double-bridge operators for the TSP. These three new operators and the four operators of Cassani and Righini (2004) are then embedded within a variable neighborhood search heuristic. As in most papers on similar routing problems, we assume that the cost function is symmetric, but our operators can easily be adapted to asymmetric instances. The remainder of this paper is organized as follows. Section 2 contains a brief description of some basic operators including those of Cassani and Righini (2004). Section 3 describes our three new operators, followed in Section 4 by the variable neighborhood search heuristic. Computational results are presented in Section 5, followed by the conclusion in Section 6.

2. Basic operators

Let T be a feasible tour. In order to execute vertex insertions and deletions on T in constant time, we represent T as doubly-linked circular list.

2.1. Definitions associated with a feasible tour T

A tour is a sequence $(S_0, ..., S_i, ..., S_{2n+1})$, where S_0 and S_{2n+1} are copies of the depot, and S_i is a pickup or a delivery vertex, otherwise. Let $S_i = x$ be a vertex of T and let $pos(x)$ be its position in T. The predecessor of x is denoted by $pred(x)$ and its successor by $succ(x)$. Thus $pred(x) = S_{i-1}$, $succ(x) = S_{i+1}$, and $pos(x) = i$. Let $p(S_i, S_j)$ be the path from S_i to S_j in T. We define a *block* $B_x(S_i, S_j)$ of T as the path $p(S_i, S_j)$, where $S_i = x^+$ and $S_j = x^-$. When it is not necessary to specify the two extreme positions S_i and S_j of a block, we omit them. A block B_x is *simple* if there are no blocks between x^+ and x^-, and it is *composed* if it fully overlaps at least one other block (Figure 1).

An overlapped block is called a *subblock*. We denote by SUB(B_x) the set of subblocks overlapped by B_x, and by SUP(B_x) the block B_y such that $B_x \in SUB(B_y)$ and $\nexists B_z \in SUB(B_y) : B_z \in SUB(B_z)$. If B_x is not overlapped, then $SUP(B_x) = \emptyset$. Note that from the feasibility of T, a composed block B_x cannot contain a pickup vertex without its corresponding delivery vertex, and vice-versa. When performing vertex insertion, we define

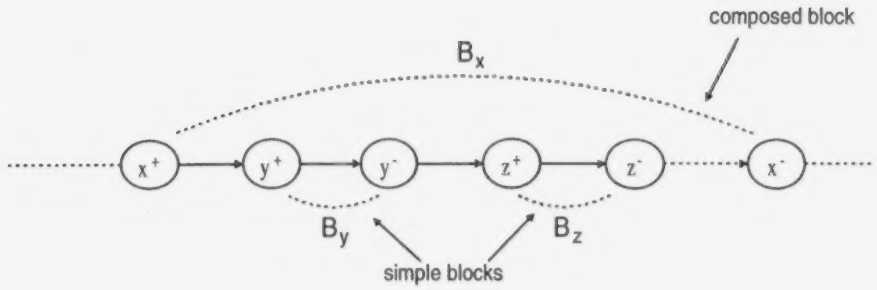

Figure 1: *The blocks B_y and B_z are simple while B_x is composed. Both B_y and B_z belong to $SUB(B_x)$.*

the *destination position* $dp(x)$ of a vertex x in T as S_i if x is inserted between S_i and S_{i+1}. The same definition also applies to a block. The extremities of a block B_x are denoted by the couple $[x^+, x^-]$. Two couples $[x^+, x^-]$ and $[y^+, y^-]$ in T are *compatible* if one of the following four *compatibility conditions* is satisfied: 1) $pos(x^+) < pos(y^+) < pos(y^-) < pos(x^-)$, 2) $pos(y^+) < pos(x^+) < pos(x^-) < pos(y^-)$, 3) $pos(x^+) < pos(x^-) < pos(y^+) < pos(y^-)$, 4) $pos(y^+) < pos(y^-) < pos(x^+) < pos(x^-)$. The first two conditions state that if one couple overlaps the other, the two couples are compatible. The last two conditions state that if two couples have no vertex in common, they are compatible. When two couples $[x^+, x^-]$ and $[y^+, y^-]$ are incompatible, we say that there exists a *cross* $crs(x, y)$ in the tour (Figure 2). Note that the presence of a cross implies that the LIFO constraint is not respected. Finally a path $p(S_i, S_j)$ is *reversible* if $x^+ \in p(S_i, S_j) \Leftrightarrow x^- \in p(S_i, S_j)$ and the LIFO constraint is respected in $p(S_i, S_j)$.

2.2. The four basic operators

We now describe four basic operators, *couple-exchange*, *block-exchange*, *relocate-block*, *relocate-couple*, introduced by Cassani and Righini (2004). These four operators rely on simple moves

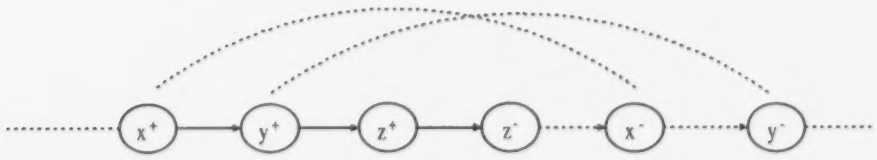

Figure 2: *The cross crs(x,y).*

which preserve the feasibility, as we now prove.

Theorem 1 *Let T be a feasible tour. Then the following statements hold: 1) the extraction of couple $[x^+, x^-]$ from T produces a new feasible tour T'; 2) the extraction of a block B_x from T produces a new feasible tour T'; 3) the insertion of a block B_x between two vertices in T produces a new feasible tour T' if $p(x^+, x^-)$ is reversible; 4) the relocation of a block B_x in T produces a new feasible tour T'; 5) the insertion of a reversible path $p(S_i, S_j)$ between two vertices in T produces a new feasible tour T'.*

Proof. 1) The proof is by contradiction. Suppose the new tour T' obtained by removing the couple $[x^+, x^-]$ from T is infeasible. This means that there is at least a cross $crs(w, z)$ in T'. Since $[x^+, x^-] \notin T'$, it follows that $w \neq x$ and $z \neq x$. Moreover, we know that the removal of $[x^+, x^-]$ from T does not change the order relation among the positions of the remaining vertices. Hence if requests w and z produce a cross in T', this cross is also in T and T is thus infeasible. 2) The removal of a block B_x from T can be executed by removing one by one all couples $[y^+, y^-]$ belonging to B_x, and then removing the couple $[x^+, x^-]$. From case 1, the extraction of a couple always yields a feasible tour and thus the final tour produced by removal of block B_x is also feasible. 3) The proof is by contradiction. Suppose that the new tour T' obtained by inserting B_x in T is infeasible. This implies that there is at least a cross $crs(w, z)$ in T'. Since B_x contains no vertices of T, from the definition of a cross we know that the insertion of this block into the tour cannot create crosses with the vertices of T. This implies that $crs(w, z)$ is either in B_x or in T. However, by hypothesis, we know that $p(x^+, x^-)$ is reversible and thus there are no crosses in B_x. Therefore, if there is a cross in T' this cross in also in T, and T is thus infeasible. 4) The relocation operation of a block B_x in T can be executed by removing this block from T and then reintroducing it into the destination position, obtaining the new tour T'. From cases 2 and 3, T' must be feasible. 5) The insertion of a reversible path can be executed by inserting, one by one, its non-overlapped blocks. From cases 3 and 4, we know that this operation always yields feasible tours and thus the final tour T' will also be feasible. □

2.2.1. The *couple-exchange* operator

The *couple-exchange* operator selects two couples $[x^+, x^-]$ and $[y^+, y^-]$ of T, swaps the positions of x^+ and y^+ and of x^- and y^-, and computes the length of the resulting tour. The operator repeats this operation for all $x, y \in R$, with $x \neq y$, and implements the best swap

if it improves upon T (Figure 3 a,b). By using appropriate pointers, this operator runs in $O(n^2)$ time.

2.2.2. The *block-exchange* operator

The *block-exchange* operator is similar to the previous one with the only difference that the swap is applied to whole blocks rather than to their extremities. For each couple of blocks B_x and B_y in T, such that $B_x \notin SUB(B_y)$ and $B_y \notin SUB(B_x)$, *block-exchange* swaps B_x and B_y and computes the length of the resulting tour. It then implements the best swap if the resulting tour improves upon T (Figure 3c). From cases 2 and 3 of Theorem 1, the new tour is feasible. Again this operator requires $O(n^2)$ time.

2.2.3. The *relocate-block* operator

The *relocate-block* operator selects a block B_x in T and relocates it in a different position. All blocks are considered for relocation and the best one is implemented if it improves upon T (Figure 4). From case 4 of Theorem 1, the new tour is feasible. This operator also runs in $O(n^2)$ time.

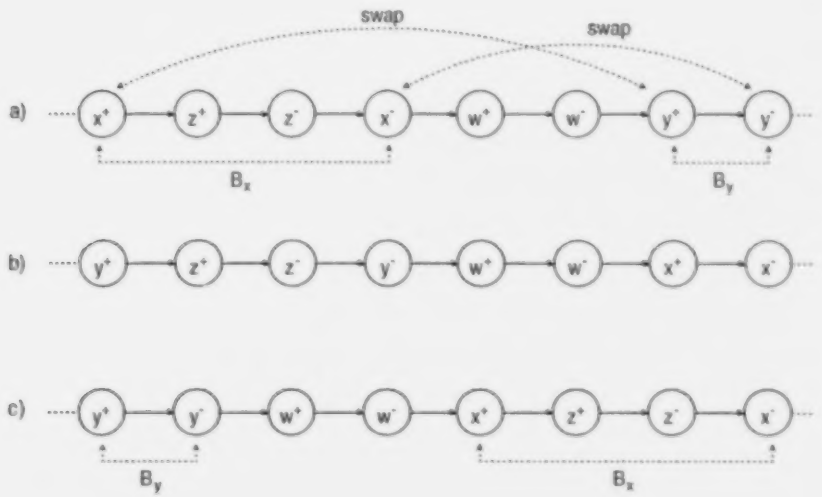

Figure 3: *a) The initial tour T. b) The new tour created by couple-exchange. c) The tour created by block-exchange.*

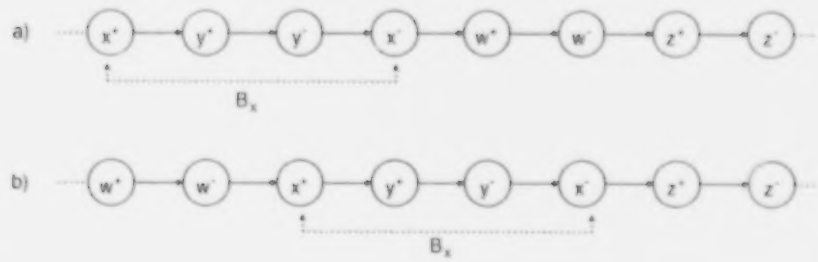

Figure 4: *a) The initial tour T. Let $dp(B_x)$ be equal to w^-. b) New tour created by relocate-block.*

2.2.4. The *relocate-couple* operator

The final operator used by Cassani and Righini (2004) is *relocate-couple*. This operator is more complicated than the others. Because our *multi-relocate* operator is based on it, we provide a detailed description of its functioning. The *relocate-couple* operator finds, for each couple $[x^+, x^-]$ of T, the best relocation position and implements the best improving relocation. This operator runs in $O(n^3)$ time. It works as follows.

For each couple $[x^+, x^-]$ of T, the vertices x^+ and x^- are first removed from T, creating a new tour T' with $2n-1$ vertices (Figure 5a). For $i = 0, \ldots, 2n-2$, set $dp(x^+) = S_i$ and introduce x^+ in its destination position. Let T'' be the new tour just created (Figure 5b). Note that $S_{i+1} = x^+$ in T''. Having inserted x^+ in T'', the operator seeks the best position S_j in $p(S_{i+1}, S_{2n-1})$ to insert x^-. Because of the LIFO constraint, not all positions in $p(S_{i+1}, S_{2n-1})$ are feasible. Therefore, the operator must first identify all feasible positions for x^- and then select the best one. The position to the right of x^+ is surely feasible for x^-. For this reason, once $dp(x^+)$ has been fixed to S_i, *relocate-couple* always sets $dp(x^-) = S_{i+1}$ in T'' and computes the length of the new tour just obtained. In order to find the other feasible destination positions in $p(S_{i+2}, S_{2n-1})$, the operator proceeds along this path, and for each vertex S_j encountered, checks whether S_j is a pickup or a delivery vertex. If $S_j = y^+$, then the operator jumps block B_y and continues the search starting from y^-. This is because if x^- is inserted in any block $B_y \in p(S_{i+2}, S_{2n-1})$, it produces the cross $crs(x, y)$ which leads to a LIFO constraint violation. If $S_j = y^-$ two cases are possible:

1) $pos(y^+) > pos(x^+)$ in T''. In this case the operator has reached vertex y^- by jumping from y^+ in the previous iteration. Hence S_j is a feasible destination position and the operator inserts x^- to the right of S_j and computes the cost of the new tour.

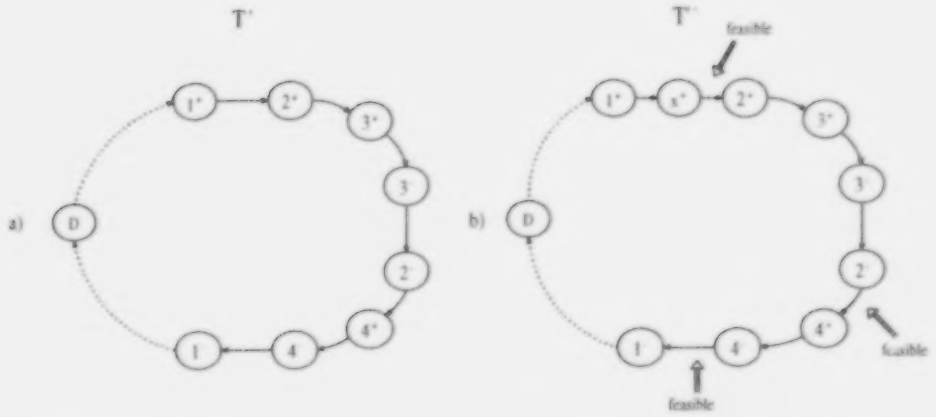

Figure 5: *a) The tour without the couple $[x^+, x^-]$. b) The vertex x^+ is inserted between 1^+ and 2^+. The feasible destination positions for x^- are x^+, 2^- and 4^-. The insertion of x^- in B_2 and B_3 produces the crosses $crs(x,2)$ and $crs(x,3)$, respectively. Note that there are no feasible positions for x^- after 1^-. Indeed introducing x^- after 1^- always creates the cross crs(1,x) because x^+ is in the block B_1.*

2) $pos(y^+) < pos(x^+)$ in T''. In this case, the search terminates because there are no other feasible destination positions for x^- after S_{j-1}. Indeed the introduction of x^- after S_j creates the cross $crs(x,y)$, which yields an infeasible tour. Using a stack, it is possible to check in constant time whether a position is feasible for x^-.

At the end the best exchange identified is implemented if it improves tour T. Because we use a doubly-linked list to represent T, it is possible to execute various changes on T in constant time. The time needed to find the best positions for each couple $[x^+, x^-]$ is $O(n^2)$ and this operation is repeated n times, once for each couple. Hence the complexity of this operator is $O(n^3)$.

3. New operators

In this section we introduce two new operators for the TSPPDL: *multi-relocate* and *2-opt-L*. We also introduce two perturbation operators used within our VNS heuristic: *double-bridge* and *shake*.

3.1. The *multi-relocate* operator

We first describe our *multi-relocate* operator, derived from the *relocate-couple* operator. As explained in the previous section *relocate-couple* computes for each couple $[x^+, x^-]$ in T the best positions to relocate x^+ and x^-. However, this information is saved only if $[x^+, x^-]$ is the best couple to relocate, and the operator relocates only the best couple identified. This implies that all the information computed by *relocate-couple* about the other couples is lost after relocating the best one. It may pay, however, to save this information for further use. The idea behind the *multi-relocate* operator is to save in a queue every couple $[x^+, x^-]$ whose relocation produces a new better tour, to relocate the best couple identified, and then to attempt to relocate as many couples as possible from the queue in the new tour. Note that when *multi-relocate* relocates only the best couple, this tour is exactly the same as the tour produced by *relocate-couple*.

We can divide the *multi-relocate* operator in two phases. In the first phase, this operator works like *relocate-couple*, the only difference being the construction of the queue. This first phase ends with the relocation of the best couple. Before describing the second phase, we discuss some issues associated with the relocation of several couples in T. Let $impr([x^+, x^-])$ be the improvement computed by *multi-relocate* during the first phase for the relocation of couple $[x^+, x^-]$ in T. All the information computed during the first phase, like the improvement associated with each couple and the destinations of its extremities, relates to the tour T. When *multi-relocate* relocates the best couple in T it produces a new tour T_1 for which some of the previous information is no longer correct. The first problem concerns the improvement value associated with each couple. This problem arises because T_1 may contain vertices whose insertion or extraction cost is changed. Moreover recomputing the improvement of each couple after each relocation is computationally prohibitive. The second problem concerns the need to maintain the feasibility of the tour after each relocation, i.e., ensuring that no crosses are created. In order to create a practically usable operator, we force *multi-relocate* to satisfy three conditions: 1) for each couple relocated in the new tour, the improvement produced has to be exactly the same as computed by *multi-relocate* on T in the first phase; 2) the tour produced after each relocation must remain feasible; 3) the overall time complexity has to be the same as that of *relocate-couple*.

The idea behind the first condition is to mark as *non-removable* all couples in the queue whose improvement cost has changed because of the last relocation. Since *multi-relocate*

relocates only the removable couples in the queue and the improvement value of these has not changed, condition 1 will be satisfied at each relocation. The problem is now to find a quick way to determinate whether a couple is removable or not.

Observation 1 *Given a tour T_k, obtained by relocating k couples in T, and a couple $[y^+, y^-]$, if the neighbors of y^+ and y^- in T and T_k are the same and the positions where y^+ and y^- will be relocated in T_k will have the same neighbors as in T, then the improvement produced to T_k associated with the relocation of $[y^+, y^-]$ is the same as $impr([y^+, y^-])$.*

We know that if a couple satisfies the conditions introduced in Observation 1, then this couple is removable. To check this we assign two flags, $removable(y)$ and $next_available(y)$, to each vertex y. The first flag indicates whether the vertex can be relocated and the second whether it is possible to insert a vertex between y and $succ(y)$. To see how these flags are used, let $[y^+, y^-]$ be the couple considered for a move in the current tour T_i, and let T_{i+1} be the new tour produced by this relocation. The operator sets the flag *removable* of $pred(y^+)$, $succ(y^+)$, $pred(y^-)$ and $succ(y^-)$ to FALSE. Indeed, after the relocation of $[y^+, y^-]$, the extraction cost of these vertices in T_{i+1} is changed and so is the improvement value of the couple associated with these nodes. In addition, the flag *next_available* of $pred(y^+)$ and $pred(y^-)$ is set to FALSE because the insertion cost of eventual vertices in these two positions in T_{i+1} will be different from the previous cost in T_i (Figure 6a). The operator then relocates $[y^+, y^-]$ and sets the *removable* flags of these vertices to FALSE because these have just been relocated. The same applies to the *removable* flags of new neighbors of y^+ and y^- in T_{i+1} (Figure 6b). Finally, *multi-relocate* sets to FALSE the *next_available* flag of vertices $pred(y^+)$, y^+, $pred(y^-)$ and y^-. Using these flags, it is possible to determine in constant time whether a couple $[y^+, y^-]$ is removable. Indeed, it is sufficient to check whether the *removable* flags of y^+ and y^- and the *next_available* flags of $dp(y^+)$ and $dp(y^-)$ take the value TRUE.

To see whether the second condition is satisfied, it is necessary to ensure that relocations do not produce crosses. The operator can check the compatibility of two couples in constant time. However, given a couple $[y^+, y^-]$, determining whether this couple satisfies condition 2 requires checking the compatibility between $[y^+, y^-]$ and all other couples already relocated, which cannot be achieved in constant time. At this point we can describe the second phase of *multi-relocate*: after the relocation of the best couple, the operator verifies whether each

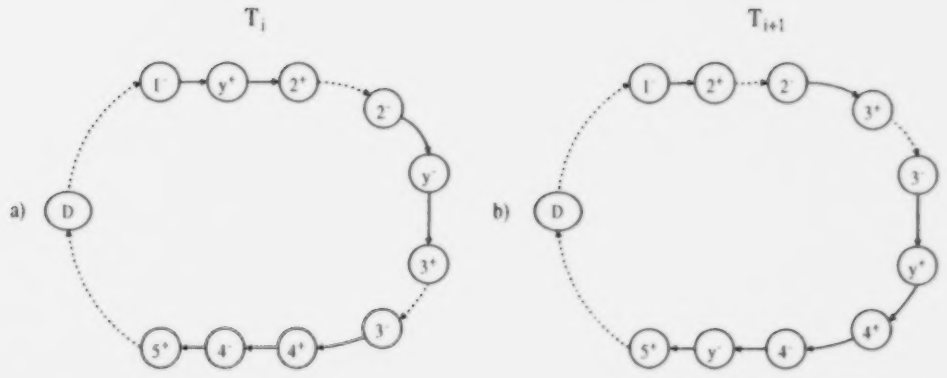

Figure 6: *a) The tour T_i. The couple to move is $[y^+, y^-]$. The flags removable(1^-), removable(2^+), removable(2^-) and removable(3^+) are set to FALSE as well as next_available(1^-) and next_available(2^-). b) The couple is relocated. The destination positions chosen are 3^- and 4^-. The flags removable(3^-), removable(4^+), removable(4^-), removable(5^+), removable(y^+) and removable(y^-) are validated to FALSE as well as next_available(3^-), next_available(4^-), next_available(y^+), next_available(y^-).*

couple in the queue satisfies conditions 1 and 2. If this is the case the couple is relocated, otherwise it is rejected.

Finally, it remains to be shown that the complexity of *multi-relocate* is $O(n^3)$. The first phase of the operator executes the same operations as *relocate-couple* and creates the queue. The complexity of this phase is $O(n^3)$. In the second phase the operator reads the queue, which contains $O(n)$ couples, and checks conditions 1 and 2 for each couple. Determining whether a couple $[y^+, y^-]$ satisfies condition 1 can be achieved in constant time because it is sufficient to verify a constant number of flags. The second check is more expensive. Indeed, the operator must verify the compatibility of $[y^+, y^-]$ with each of the $O(n)$ couples already relocated. The complexity of the second phase is therefore $O(n^2)$, which yields an overall complexity of $O(n^3)$. Note that after the relocation of a couple, it is necessary to recompute in $O(n)$ time the positions of the vertices in the new tour in order to be able to check the compatibility conditions. It is, however, possible to reduce the cost of this operation by using a simple observation. To determine whether two couples $[x^+, x^-]$ and $[y^+, y^-]$ are compatible in T, it is not necessary to know the exact positions of their extremities in the tour, but only their visiting order. In other words, if one of following conditions holds then the two couples are compatible: y^- precedes x^+; y^+ follows x^-; x^+ precedes y^+ and y^- precedes x^-; y^+ precedes x^+ and x^- precedes y^-.

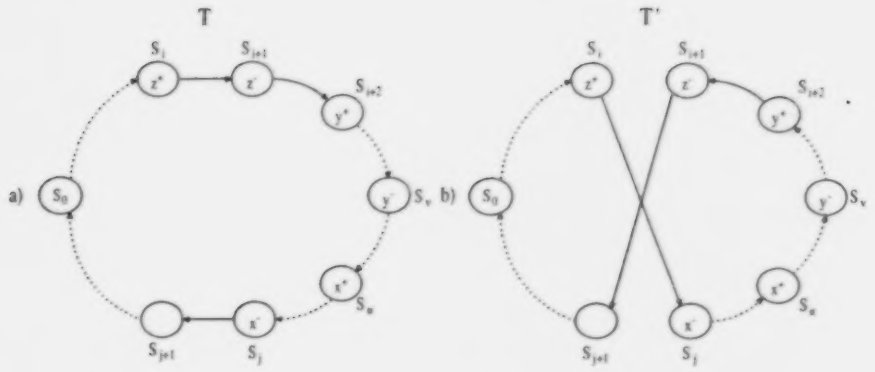

Figure 7: *a) A feasible tour. b) A new tour created by the 2-opt operator. The inversion of* $p(S_{i+1}, S_j)$ *produces an infeasible tour because* $pos(x^-) < pos(x^+)$ *and* $pos(y^-) < pos(y^+)$.

3.2. The *2-opt-L* operator

We now show how to adapt the 2-opt operator to the TSPPDL. This operator involves the substitution of two arcs, (S_i, S_{i+1}) and (S_j, S_{j+1}), with two other arcs, (S_i, S_j) and (S_{i+1}, S_{j+1}), and the reversal of path $p(S_{i+1}, S_j)$. The new tour, produced after this 2-exchange is $T' = (p(S_0, S_i) \cdot p(S_{i+1}, S_j)^R \cdot p(S_{j+1}, S_{2n+1}))$ (Figure 7). Because $p(S_{i+1}, S_j)$ and $p(S_{i+1}, S_j)^R$ have the same length, the cost of T' will be less than the cost of T if and only if

$$c(S_i, S_{i+1}) + c(S_j, S_{j+1}) > c(S_i, S_j) + c(S_{i+1}, S_{j+1}). \tag{1}$$

The 2-opt operator considers the $O(n^2)$ 2-exchanges associated with a given tour and implements the best one. It is easy to see that if T is a feasible tour for the TSPPDL, the application of a 2-exchange will produce an infeasible tour T' because the precedence constraint will be violated (Figure 7b).

We must therefore create an operator that preserves the precedence and LIFO constraints. Given a path $p \in T$, the idea is to reverse the visiting order of blocks in p, instead of single vertices. This is achieved through the REVERSE procedure introduced in Section 3.2.1. However this function works only on reversible paths and before invoking it we have to check in $O(n)$ time whether p is reversible. Since there are $O(n^2)$ possible 2-exchanges in the tour, the feasibility check for each of them increases the complexity of the operator to $O(n^3)$. Psaraftis (1983) and Savelsbergh (1990) have introduced strategies to check the feasibility of a k-exchange without increasing the complexity of the operator in the presence of precedence constraints. We apply these strategies to our problem in which we must also ensure that the

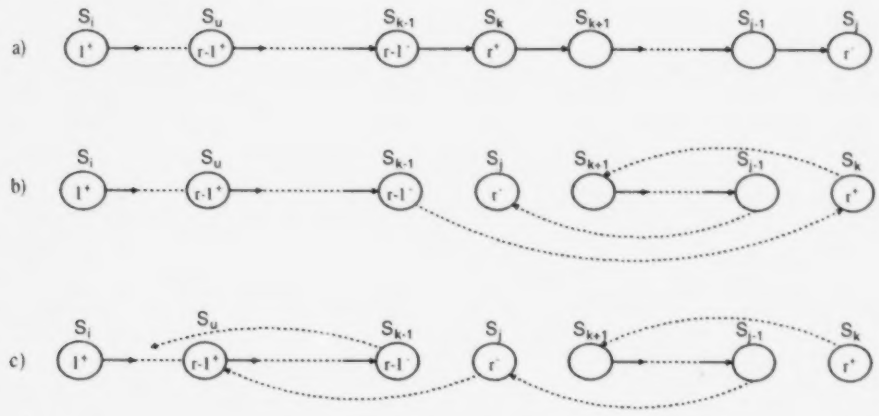

Figure 8: *a) The reversible path $p(S_i, S_j)$. The blocks B_1, B_{r-1} and B_r are the first, the second last and the last block of the path, respectively. b) The extremities of B_r are swapped.(c) REVERSE adds the arc between r^- and $(r-1)^+$. B_{r-1} becomes the last block visited.*

LIFO constraint is satisfied. The idea is to divide the operator in two phases. The first one computes all feasible 2-exchanges in the current tour, and the second one considers only the feasible ones in order to find the best exchange. In Section 3.2.2 we introduce the CHECK procedure which computes in $O(n^2)$ time all reversible paths and feasible 2-exchanges in a given feasible tour.

3.2.1. The REVERSE procedure

Inverting the visiting order of vertices of a reversible path $p(S_i, S_j)$ produces a new path $p(S_i, S_j)^R$ in which the precedence and LIFO constraints are violated since the delivery vertices come before the associated pickups. To solve this problem it is sufficient to reverse the visiting order of the blocks of $p(S_i, S_j)$ rather than that of the vertices. We know that a block belonging to a feasible tour is reversible. Moreover, from case 4 of Theorem 1 we know that each permutation of the blocks of a reversible path produces a new reversible path. Therefore in $p(S_i, S_j)^R$ the precedence and LIFO constraints hold. We now summarize the three steps of the REVERSE procedure applied to $p(S_i, S_j)$ (Figure 8):

Step 1. Swap the extremities of the last non-overlapped block in $p(S_i, S_j)$.

Step 2. Let B_q be the last block visited and let B_p be the block preceding B_q in $p(S_i, S_j)$, with $SUP(B_p) = \emptyset$. Then set $succ(q^-) = p^+$ and B_p as the last visited block.

Step 3. Repeat Step 2 until the last delivery vertex visited is the delivery of the first block in $p(S_i, S_j)$.

REVERSE also computes the difference between the costs of the removed and added edges and returns the length of $p(S_i, S_j)^R$. Note that even in the symmetric case this computation cannot be avoided because $p(S_i, S_j)^R$ contains new arcs with respect to $p(S_i, S_j)$. Unfortunately, this computation increases the complexity of the *2-opt-L* operator. It is easy to see that REVERSE runs in $O(n)$ time.

3.2.2. The CHECK procedure

The CHECK procedure identifies all reversible paths in a feasible tour T in order to execute feasible 2-exchanges. For each S_i and S_j, such that $1 \leq i \leq 2n-1$ and $i+1 \leq j \leq 2n$, the procedure determines whether $p(S_i, S_j)$ is reversible. The results of these computations are saved in a $2n \times 2n$ matrix, called REV, in which $REV[i,j]$ = TRUE if and only if the path $p(S_i, S_j)$ is reversible. To perform this computation the procedure uses a stack and a counter, *top*, representing the first free position in the stack. Given a path $p(S_i, S_j)$, the function scans the vertices from S_i to S_j. CHECK puts each pickup vertex x^+ on top of the stack and increments *top* by one. When a delivery vertex x^- is encountered, the procedure checks whether the pickup node x^+ is in position $top-1$ on the stack. If this is not the case, then $x^+ \notin p(S_i, S_j)$ and $p(S_i, S_j)$ is not reversible. Otherwise, CHECK removes x^+ from the stack, decrements *top* by one and proceeds to the next vertex. The path $p(S_i, S_j)$ will be reversible if and only if vertex S_j is reached with *top* equal to zero. This procedure can be accelerated as follows. A path starting with a delivery vertex or finishing with a pickup vertex cannot be reversible. We can therefore reduce REV to an $n \times n$ matrix, where the row indices are associated with the n pickup vertices in T, and the column indices with the n delivery vertices. The time complexity of the CHECK procedure is $O(n^2)$. The initialization of REV to FALSE requires $O(n^2)$ time. The operations executed on the stack run in constant time. The time required to validate a row of REV is equal to $O(n)$, and operation is repeated n times. The total time needed to validate REV is therefore $O(n^2)$.

3.2.3. The operator

The *2-opt-L* operator first applies the CHECK function to a feasible tour T to identify all feasible 2-exchanges. For each couple of vertices S_i and S_j such that $1 \leq i \leq 2n-3$ and

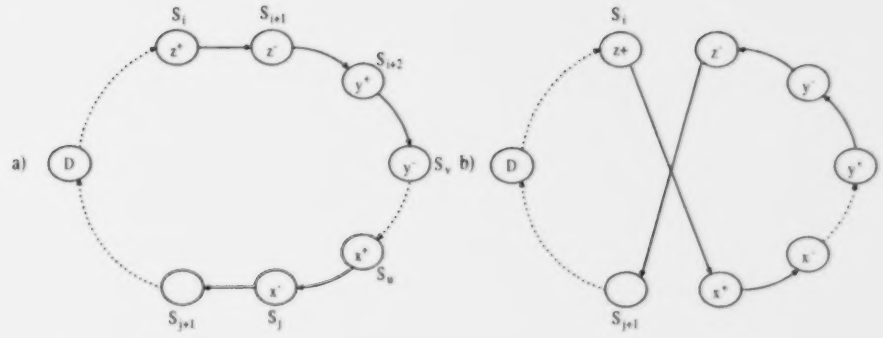

Figure 9: *a) The feasible tour T. b) The resulting tour produced by* 2-opt-L *on* S_i *and* S_j.

$i+2 < j \leq 2n$, *2-opt-L* executes one of following operations, according to the values of S_i and S_{i+1}:

1) $S_i = z^+$ and $S_{i+1} = z^-$ (Figure 9a). If $SUP(B_z) = \emptyset$ or $B_z \in SUB(B_k)$ and $pos(k^-) \geq j+1$, *2-opt-L* checks through the matrix *REV* whether $p(S_{i+2}, S_j)$ is reversible. If not, the operator proceeds to the next j. Otherwise, let $B_y(S_{i+2}, S_v)$ and $B_x(S_u, S_j)$ be the first and last block of $p(S_{i+2}, S_j)$, with $SUP(B_x) = SUP(B_y) = \emptyset$, respectively. The operator applies the REVERSE function on $p(S_{i+2}, S_j)$ to obtain the path $p(S_{i+2}, S_j)^R$. It then removes the arcs (S_i, S_{i+1}), (S_j, S_{j+1}), and (z^-, y^+) and adds the arcs (z^+, x^+), (y^-, z^-) and (z^-, S_{j+1}) to create a new feasible tour (Figure 9b). If $B_z \in SUB(B_k)$ and $pos(k^-) \leq j$, then $p(S_{i+2}, S_j)$ is not reversible for any $j > pos(k^-)$ because $pos(k^+) < i+2$. The operator therefore proceeds to the next i.

2) $B_z(S_i, S_k)$ and $i+1 < k < j$ (Figure 10). In this case path $p(S_{i+2}, S_j)$ is not reversible because $z^+ \notin p(S_{i+2}, S_j)$. We therefore focus our attention on the path $p(S_{k+1}, S_j)$. If $p(S_{k+1}, S_j)$ is not reversible the operator proceeds to the next j. Otherwise it computes $p(S_{k+1}, S_j)^R$ and introduces it between S_i and S_{i+1} (Figure 10b).

3) $B_z(S_i, S_k)$ and $k \geq j$. In this case the operator proceeds to the next j and restarts because it is preferable not to change the internal structure of composed blocks B_z.

4) $S_i = z^-$. The operator checks whether $p(S_{i+2}, S_j)$ is reversible. If not it proceeds to the next j. Otherwise, it constructs $p(S_{i+2}, S_j)^R$ and inserts it between S_i and S_{i+1} as in case 1.

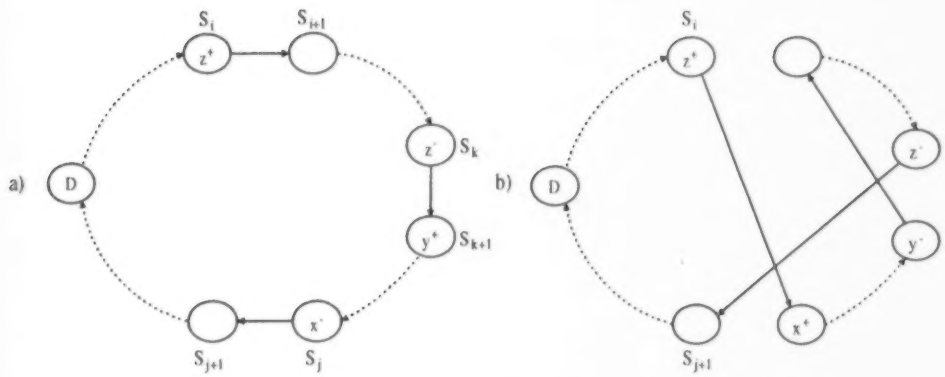

Figure 10: *a) The feasible tour T. b) The resulting tour produced by* 2-opt-L *by replacing the arcs (S_i, S_{i+1}) and (S_j, S_{j+1}). In this case $p(S_{i+2}, S_j)$ is only partially reversed because $z^- \in p(S_{i+2}, S_j)$.*

During the computations, *2-opt-L* saves the best 2-exchange identified and implements it if it improves upon T. The complexity of the *2-opt-L* operator is computed as follows. The $O(n^2)$ CHECK function is called once at the beginning. For fixed S_i and S_j, REVERSE runs in a time proportional to the length of $p(S_{i+2}, S_j)$ and has an $O(n)$ complexity. (Note that since the REVERSE function returns the length of the reversed path, *2-opt-L* can compute in constant time the eventual improvement yielded by the 2-exchange). The change of three arcs requires constant time and thus the complexity of each 2-exchange is $O(n)$. Since the number of possible couplings, S_i and S_j, is equal to $O(n^2)$, the overall complexity of *2-opt-L* is $O(n^3)$.

3.3. Perturbation operators

We now introduce two operators used by our heuristic to perturb solutions. Other perturbation heuristics that could be adapted to the TSPPDL can be found in Renaud, Boctor and Laporte (2002).

3.3.1. The *double-bridge* operator

The *double-bridge* operator perturbs a local optimum identified by a local search algorithm. The classical *double-bridge* operator, introduced by Lin and Kernighan (1973), selects four breakpoints b_1, b_2, b_3, b_4 defining four paths A,B,C,D, in T and constructs a new tour T_1 by replacing the arcs $(b_1, succ(b_1))$, $(b_2, succ(b_2))$, $(b_3, succ(b_3))$, $(b_4, succ(b_4))$ with $(b_1, succ(b_3))$, $(b_4, succ(b_2))$, $(b_3, succ(b_1))$, $(b_2, succ(b_4))$ (Figure 11). In order to simplify the description

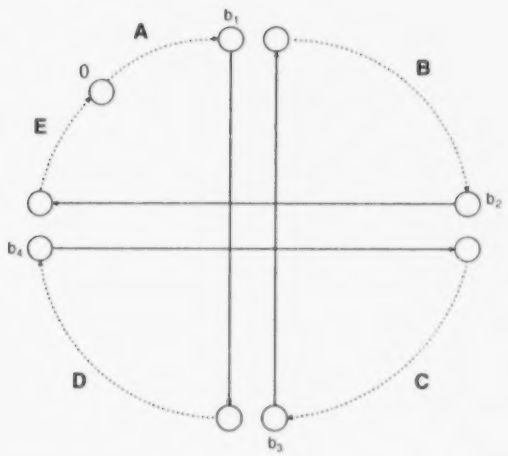

Figure 11: *The double bridge operator. The tour is broken in four points* b_1, b_2, b_3, b_4.

we divide the path containing vertex 0 into two paths: A, from $succ(0)$ to b_1, and E, from $succ(b_4)$ to 0. This last path does not exist if $b_4 = pred(0)$. The tour $T = (A \cdot B \cdot C \cdot D \cdot E)$ is then replaced with the tour $T_1 = (A \cdot D \cdot C \cdot B \cdot E)$.

Because of the precedence and LIFO constraints, T_1 may not be feasible for the TSPPDL. One must therefore select the four paths A, B, C and D to ensure the feasibility of T_1. An easy way to solve this problem is to restrict the search to reversible paths. However, this will rarely be possible and it is necessary to allow the creation of non-reversible paths, which complicates the operator. To describe how *double-bridge* selects the breakpoints, we introduce the following theorem.

Theorem 2 *Let $p(S_i, S_j)$ be a reversible path of a tour T and let $i < k < j$. If $p(S_i, S_k)$ is reversible then $p(S_{k+1}, S_j)$ is also reversible.*

Proof. Without loss of generality, let $S_i = x^+$, $S_k = y^-$ and $S_j = z^-$. The proof is by contradiction. Suppose that $p(S_i, S_k)$ is reversible and $p(S_{k+1}, S_j)$ is not. This implies that at least one of the following two cases holds: 1) there is at least one pickup vertex $w^+ \in p(S_{k+1}, S_j)$ for which the corresponding delivery vertex is not in this path. This means that T contains the cross $crs(w, z)$ and thus $p(S_i, S_j)$ is not reversible; 2) there is at least one delivery vertex $w^- \in p(S_{k+1}, S_j)$ for which the corresponding pickup vertex is not in this path. Here we have to consider the following two subcases: $w^+ \in p(S_i, S_k)$ and $w^+ \in p(S_0, S_{i-1})$. In the

first subcase, since $p(S_i, S_k)$ contains a pickup vertex without its delivery vertex, the path is not reversible. In the second subcase $p(S_i, S_j)$ contains a delivery vertex without its pickup vertex and thus $p(S_i, S_j)$ is not reversible. □

The idea behind the selection of the breakpoints is to select them randomly in different quarters of T, when possible. The first breakpoint b_1 is selected randomly among the delivery vertices, followed by a pickup vertex between $succ(0)$ and $S_{\lfloor n/2 \rfloor}$. If there is no such delivery vertex, then the first delivery vertex found after $S_{\lfloor n/2 \rfloor}$ and followed by a pickup vertex is selected. Having selected b_1, the selection of the remaining three breakpoints is made according to one of the following cases:

1) A is reversible. In this case whatever the selection of the remaining three pieces, the vertices of A never violate the LIFO constraint in T_1. For this reason the construction of B, C and D depends on the selection of b_2. The operator randomly selects a delivery vertex b_2 in $p(succ(succ(b_1)), S_n)$. If there are no deliveries in this path, then the operator selects the first delivery vertex encountered after S_n. For a given breakpoint b_2 we consider the following two subcases: 1a) If B is reversible, it never violates the LIFO constraint in T_1. However the selection of b_3 cannot be totally random. Indeed, since D comes before

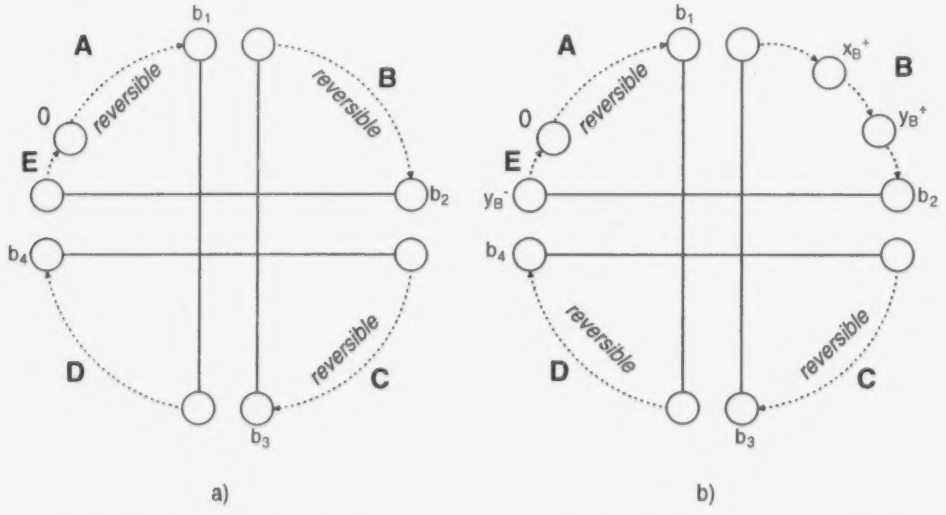

Figure 12: *a) A and B are reversible. Double-bridge selects b_3 randomly in $p(succ(succ(b_2)), S_{2n-3})$ and so that C is reversible. b) B is not reversible. Double-bridge sets $succ(b_4)$ to y_B^-, inserting all vertices of INF_D_B in E. It then selects b_3 so that C and D are reversible.*

C in T_1, if C is not reversible then T_1 can be infeasible. Feasibility occurs only if each pickup vertex in C has its delivery vertex in C or E. To ensure feasibility, *double-bridge* randomly selects the breakpoint b_3 among all delivery vertices in $p(succ(succ(b_2)), S_{2n-3})$ so that C is reversible (Figure 12a). Note that in this case one can randomly select b_4, in $p(succ(succ(b_3)), pred(2n+1))$. Indeed *double-bridge* inserts two reversible paths, B and C, between b_4 and its successor, and we know from Theorem 1 that this operation never violates the precedence and LIFO constraints. The search for b_3 is made as far as S_{2n-3}, in order to increase the probability of finding a reversible path C and to leave at least two vertices in the remainder of T for b_4.

1b) B is not reversible. Let INF_P_B be the set of pickup vertices of B for which the delivery vertex is not in B, and let INF_D_B be the set of delivery vertices with pickups belonging to INF_P_B. Let x_B^+ and y_B^+ be the first and last pickup vertices in B such that $x_B^+, y_B^+ \in INF_P_B$ (Figure 12b). If $|INF_P_B| = 1$, then $x_B^+ = y_B^+$. One cannot leave any delivery vertices of INF_D_B in C or D because this would yield a violation of the precedence or LIFO constraints. For this reason *double-bridge* sets $succ(b_4) = y_B^-$, thus fixing the fourth breakpoint, and inserts all vertices of INF_D_B in E . Moreover since D comes before C in T_1, one must ensure that both pieces are reversible. Thus *double-bridge* chooses b_3 randomly in $p(succ(succ(b_2), pred(pred(b_4)))$ so that $p(succ(b_2), b_3)$ is reversible. Note that from Theorem 2, if $p(succ(b_2), b_4)$ is reversible and b_3 is fixed like above, the $p(succ(b_3), b_4)$ will be reversible.

2) A is not reversible. From the feasibility of T we know that A contains at least one pickup vertex whose delivery vertex comes after b_1. Let INF_P_A, INF_D_A, x_A^+ and y_A^+ be defined as in case 1b. To choose the other three breakpoints *double-bridge* must determine in which of the three paths B, C or D it will insert the vertices of INF_D_A. This choice could be made arbitrarily, but the operator considers each possibility separately in order to increase the likelihood of producing a feasible tour T_1 (see Carrabs (2005) for a detailed explanation of each case).

We now determine the complexity of this operator. The search for the four breakpoints requires the scanning of the whole tour for a total complexity of $O(n)$, and this operation is repeated a constant number of times, once for each case described above. The reversibility checks, executed during each case, are done in constant time using the REV matrix. However, this implies that before applying the operator we have to call the CHECK function,

for an additional computation time of $O(n^2)$. Having selected the four breakpoints in $O(n)$ time, the operator constructs T_1 in constant time by changing a constant number of pointers. The *double-bridge* operator therefore runs in $O(n^2)$ time.

3.3.2. The *shake* operator

While the *double-bridge* operator is rather efficient at perturbing solutions, it sometimes fails and another operator, called the *shake* operator, is then applied. It randomly selects one of the following four operators: *couple-exchange*, *block-exchange*, *relocate-couple*, *relocate-block*. According to the type of operator selected, *shake* randomly selects a couple or a block for a random relocation or swap in the tour. Because of the randomness involved, *shake* can be applied several times without running the risk of cycling. To prevent our local search algorithm from undoing the work of *shake*, the couple or block moved by this operator is declared *tabu* for a number of iterations.

4. Variable neighborhood search heuristic

Variable neighborhood search (VNS) was introduced by Mladenović and Hansen (1997). The idea is to apply a perturbation to the current neighborhood operator at a local minimum. The perturbation enables the search to reach a solution that could not have been reached by the current local search mechanism and thus yields a broader exploration of the search space. Our VNS heuristic is summarized in Figure 13.

Procedure: *VNS*
1: $s \leftarrow$ *Generate_starting_tour*;
2: $s_1 \leftarrow$ *LocalSearch*(s);
3: **while** termination condition is false **do**
4: $\quad s_2 \leftarrow$ *Perturbation*(s_1);
5: $\quad s_3 \leftarrow$ *LocalSearch*(s_2);
6: $\quad s_1 \leftarrow$ *AcceptanceCriterion*(s^*, s_3);
7: **end while**

Figure 13: Variable neighborhood search procedure

Local search is first applied to a starting solution s until a local minimum s_1 is encountered. A loop made up of three operations is then executed until a termination condition is met. The first operation perturbs the current local optimum s_1 to obtain a new solution s_2. The second operation reapplies the local search algorithm on s_2 to obtain a locally optimal

Heuristic	*Description*
CI	***Cheapest Insertion:*** At each iteration the heuristic computes for each couple $[x^+, x^-]$ not yet selected the best position where to insert it into the tour. It then inserts the couple whose insertion minimizes the length of the current tour.
LI	***Largest Insertion:*** Like the CI heuristic but at each iteration the couple whose insertion maximizes the length of current tour is selected. The first couple is selected as in the CI heuristic.
FI	***Fastest Insertion:*** In the first iteration the heuristic selects the couple whose insertion maximizes the length of the current tour. In the following iterations for each couple not yet selected, the heuristic computes the distances from the two vertices of the couple and all vertices of the current tour. It selects the couple with the maximal minimum distance and inserts it in its best position in the tour.
NI	***Nearest Insertion:*** Like the FI heuristic but the couple selected is the one closest to the nodes of current tour.
NN	***Nearest Neighbor:*** At each iteration insert the nearest neighbor of the last vertex inserted. The insertion starts with a pickup vertex.
RAN_NN	***Randomized Nearest Neighbor:*** For a fixed parameter λ, at each iteration one of the λ nearest vertices to the last one is randomly selected and inserted.
REV_NN	***Reversed Nearest Neighbor:*** Similar to the NN heuristic but the insertion starts with a delivery vertex.
REV_RAN_NN	***Reversed Randomized Nearest Neighbor:*** Similar to RAN_NN, but with a different random selection of vertices.

Figure 14: *List of heuristics used to generate the starting tour.*

solution s_3. Finally the last operation determines whether the next iteration will start from the current solution s_3 or from the best one s^*.

In our VNS implementation, the starting solution is obtained by using one of the eight construction heuristics proposed by Cassani and Righini (2004). These heuristics are described in Figure 14. The neighborhoods are defined by the following operators: *couple-exchange*, *block-exchange*, *relocate-block*, *2-opt-L* and *multi-relocate*, which are applied in a predefined order. Whenever an operator produces a new improving solution, the search restarts from the first operator. Otherwise, the next operator is applied. The heuristic stops when no operator improves the current solution. We have created two VNS heuristics using a different ordering of the operators. The first one, VNS_1, applies the operators in the following order: *couple-exchange*, *block-exchange*, *relocate-block*, *2-opt-L*, *multi-relocate*. The second one, VNS_2, applies the operators in the following order: *2-opt-L*, *couple-exchange*, *block-exchange*, *relocate-block*, *multi-relocate*. VNS_2 is used in line 5 of Figure 13. We have used two different heuristics because of the type of solutions produced by *2-opt-L*. Indeed, the solutions produced by this operator do not usually significantly differ in terms of cost, but

may have very different structures because of the path reversion. This implies that applying VNS_2 to the starting tour can produce random jumps, especially during the first iterations. In order to avoid this we apply VNS_1 to the starting tour. However, the solutions produced by *2-opt-L* can turn out to be useful after the perturbation of the local minimum. Indeed, the repeated application of the first three operators of VNS_1 on a solution perturbed by the *shake* operation can easily bring the search back to the last local optimum found. If this happens, then the remaining two operators become useless because they cannot improve this local optimum. The application of *2-opt-L* to the perturbed solution decreases the probability of returning to the last local minimum. To further reduce this risk we also use a tabu list that saves all the operations executed by *couple-exchange*, *block-exchange*, *relocate-block* and *multi-relocate* during the intensification phase. Note that *relocate-couple* is not used in our VNS heuristic because *relocate-couple* and *multi-relocate* are two mutually exclusive operators. Both compute the best couple to relocate in the current tour.

4.1. Perturbation phase

The *perturbation phase* is implemented with the *double-bridge* operator. This operator changes only four edges of a tour and does not execute reverse operations. This implies that usually the cost of the new tour is close to the original one. Another good property of the *double-bridge* operator is that it is not easy for the other operators to undo the changes it performs on the tour, which means there is little chance of going back to the last perturbed local minimum. For this reason the operations of *double-bridge* are not inserted in the tabu list. When the operator fails, we perturb the local optimum with the *shake* operator. These moves are introduced in the tabu list and, because VNS_2 uses the same operator, we reapply the *double-bridge* operator to the tour produced by *shake* in order to reduce the likelihood of going back to the same perturbed local optimum. For the same reason *double-bridge* is applied after the *shake* in the diversification phase.

4.2. Acceptance criterion

The last part of the heuristic concerns the *acceptance criterion*. Here we have to decide whether the solution produced by VNS_2 will be accepted as a starting solution for the next iteration. Obviously, if the new tour computed by VNS_2 is better then the current best, we save it and accept this new solution. Otherwise we accept the new solution only if $current_cost - \alpha/distance \leq best_cost$, where $current_cost$ and $best_cost$ are, respectively,

```
Procedure: VNS

1: s ← Starting_solution();
2: s_1 ← VNS_1(s);
3: diversification ← 0;
4: for iteration = 0 to MAX_ITER do
5:
6:    /* perturbation phase */
7:    if s_j ← double_bridge(s_1) fails then
8:       s'_1 ← shake(s_1);
9:       s_j ← double_bridge(s'_1);
10:   end if
11:
12:   if (cost(s_j) < best_cost) then
13:      s* ← s_j
14:      best_cost ← cost(s_j);
15:      iteration ← 0;
16:   end if
17:
18:   s_k ← VNS_2(s_j);
19:
20:   if (cost(s_k) < best_cost) then
21:      s* ← s_k
22:      best_cost ← cost(s_k);
23:      iteration ← 0;
24:   else
25:      s_k ← Acceptance_criteria(s*, s_k);
26:   end if
27:   s_1 ← s_k;
28:
29:   /* diversification phase */
30:   if (iteration = MAX_ITER AND diversification = 0) then
31:      iteration ← 0;
32:      diversification ← 1;
33:      s'_1 ← shake(s_1);
34:      s_1 ← double_bridge(s'_1);
35:   end if
36: end for
```

Figure 15: Pseudocode of VNS heuristic.

the length of the tour produced by VNS_2 and the best one computed so far, *distance* is the number of different arcs between the two tours, and $\alpha = size \times iteration^2$. The idea behind this formula is the following. Since the heuristic is executed within an intensification phase, we do not want to move too far away from the current local optimum. Consequently, the larger the cost and the number of different edges between the new solution and best one, the lower is the probability of accepting this new solution. Figure 15 shows the pseudocode of our VNS heuristic.

5. Computational results

The algorithm just described was coded in C and run on a 3.4 Ghz PENTIUM 4 Processor. We compare our variable neighborhood search heuristic to the VND heuristic proposed by Cassani and Righini (2004). We first report additional details about the parameters

used in the VNS heuristic. The termination condition of the heuristic is the number of consecutive iterations without improvement in the current best solution. The *shake* operator is applied three times during the perturbation phase while the number of applications of *double-bridge* depends on instance size. More precisely, let *size* be equal to $2n + 1$: if $size < 251$ this operator is applied only once in the diversification phase, otherwise it is applied $2 + \lfloor (size - 251)/500 \rfloor$ times. Our implementation of *double-bridge* ensures that every time the operator is applied, the four selected breakpoints are different from those of the previous application. In the diversification phase *shake* is applied six times and the *double-bridge* the same number of times as in the perturbation phase, plus one. The size of the tabu list is equal to 20 if $size < 75$, and $(size \times 25)/100$, otherwise.

Test instances were created from the six files *fnl4461, brd14051, d15112, d18512, nrw1379, pr1002* of TSPLIB. In each case, seven subsets of customers were considered to yield instances containing *25, 51, 75, 101, 251, 501* and *751* vertices. For each instance size a random matching was performed among the vertices to create pickup and delivery pairs. Also, for each instance, eight different starting tours were obtained by applying the heuristics described in Figure 14. The use of different starting tours allows us to see how the quality of the starting solution affects the final results of the heuristics. We have divided the test results in two tables. In Table 1 we report the solution values computed by our VNS for each starting solution. Obviously the change of starting tour implies a different final result. In each line we indicate in bold the best value computed for that instance. The last line reports, for each starting tour, how many times the VNS has found the best solution starting from this solution. In particular we can see that the best results are obtained for the FI and RAN_NN heuristics. We have therefore selected one of these two heuristics (the RAN_NN) to produce the starting tour on which were applied both the VND heuristic of Cassani and Righini (2004) and our VNS heuristic. Table 1 shows that our heuristic is quite stable irrespective of the starting solution, the average gap between the best and worst solutions being around 2%. In contrast, the results produced by VND with the different starting tours (not reported here for the sake of brevity) show an average gap of 8%.

Table 2 gives the results obtained with the RAN_NN, VND and VNS heuristics. For each instance we report the cost of the final tour produced by the heuristics and, for VND and VNS, the CPU time in seconds. Because of the random choices made in the VNS heuristic, this algorithm is not deterministic. It is therefore executed ten times on each instance. We report average cost value and computing time over these ten executions. The

Instance	Size	*CI*	*LI*	*NI*	*FI*	*NN*	*RAN_NN*	*REV_NN*	*REV_RAN_NN*
fnl4461	25	**2168.0**	**2168.0**	**2168.0**	**2168.0**	**2168.0**	**2168.0**	**2168.0**	**2168.0**
	51	4022.8	4064.1	4048.1	4022.8	4039.7	**4020.0**	4142.1	4051.4
	75	5931.3	5768.4	5838.0	5858.1	5850.4	5865.0	5823.7	**5758.9**
	101	8807.8	8774.9	8935.4	8884.3	**8715.7**	8852.8	8782.3	8853.0
	251	29501.7	29446.1	29525.6	29594.1	29881.9	**29330.6**	29718.1	29703.2
	501	72652.5	72217.1	73040.7	72443.6	73886.0	**71208.5**	71828.1	72684.3
	751	118756.1	119315.6	118828.1	118443.2	119722.3	**118383.1**	119460.7	119226.2
brd14051	25	4682.2	4680.0	4685.6	**4678.8**	4680.6	4682.2	4685.6	4695.8
	51	**7746.3**	7864.0	7811.7	7765.5	7749.1	7763.2	7759.0	7782.4
	75	7300.7	7262.5	7269.2	7242.6	7364.5	7309.1	**7242.4**	7270.3
	101	9927.7	9865.8	9948.9	**9818.3**	9915.6	10005.2	10156.8	10216.0
	251	**23662.2**	24120.9	24131.6	24269.8	24346.4	24119.3	24152.2	23775.0
	501	52496.6	**52238.0**	53248.4	52636.9	52415.9	52806.8	52637.8	52769.2
	751	85699.8	86690.8	85922.2	86047.3	**85486.2**	86230.1	85934.5	85940.7
d15112	25	**93981.0**	94307.1	94614.2	93981.0	94297.6	93981.0	94028.3	94028.3
	51	142537.3	**142179.2**	142377.8	143752.7	146058.3	143575.2	143575.2	145614.9
	75	203073.8	203498.5	**200904.9**	203336.3	201818.6	201385.4	204391.4	203669.3
	101	274304.6	274183.3	**273181.3**	273615.9	276765.0	276876.8	274580.3	276711.9
	251	**581953.1**	588996.4	588573.1	582951.2	582872.9	589066.9	584883.0	590505.6
	501	956892.3	960192.3	960239.2	**953650.9**	954507.7	953764.5	963289.6	957983.9
	751	1354634.5	1354651.7	1363911.1	1357396.9	1343621.8	1352866.6	**1341634.8**	1360229.3
d18512	25	**4678.8**	4685.6	4686.8	4692.4	4685.6	4683.4	4679.4	4685.6
	51	7541.9	7569.5	7561.5	**7539.5**	7601.4	7565.6	7562.3	7554.8
	75	8791.7	8742.6	**8658.2**	8677.4	8669.8	8781.5	8813.7	8797.4
	101	10442.1	10390.6	10417.1	10605.4	10397.2	**10332.4**	10390.9	10404.9
	251	24551.3	**24376.9**	24894.3	24874.5	25214.8	24855.4	24931.4	24717.9
	501	51262.7	51554.3	51964.1	51350.9	51508.7	52295.6	51825.4	**51203.3**
	751	84359.8	84025.1	83787.7	84253.8	83875.0	83763.3	84420.7	**83737.6**
nrw1379	25	3193.4	3193.4	3196.2	3193.4	3193.4	3194.8	3193.4	**3192.0**
	51	5078.4	5067.7	**5055.0**	**5055.0**	5089.7	5095.0	5071.8	5086.8
	75	7034.2	7050.3	6931.5	7033.0	7079.3	**6865.1**	6952.4	6965.4
	101	10167.8	**10004.6**	10032.1	10132.5	10189.7	10197.5	10158.7	10205.6
	251	27699.9	27712.2	27925.6	**27652.5**	27782.3	27936.2	27934.4	27939.5
	501	60925.1	60928.0	60529.6	60387.2	60890.0	60584.5	60671.6	**60222.5**
	751	**104902.5**	105547.0	105182.7	105966.2	105155.0	105136.1	105400.1	105543.2
pr1002	25	**16221.0**	**16221.0**	**16221.0**	**16221.0**	**16221.0**	**16221.0**	**16221.0**	**16221.0**
	51	31151.9	31067.3	32506.4	**30936.0**	31162.9	31186.6	30983.2	31243.9
	75	47371.0	47066.1	47894.7	47401.8	47024.7	**46911.0**	47108.3	46980.3
	101	62753.9	**62527.5**	63458.6	62802.5	63610.3	63611.1	62787.3	62721.6
	251	200022.1	**198226.3**	200114.7	199210.4	201451.8	200028.5	198501.0	199296.3
	501	485685.3	482324.0	485204.5	486390.5	487077.0	485042.3	484402.2	**481850.6**
	751	815096.1	818410.3	812640.2	816053.0	**807311.8**	819197.7	818935.6	816962.1
Nb best		*8*	*8*	*5*	*9*	*5*	*9*	*4*	*7*

Table 1: Test results of VNS applying the eight different starting tours.

last column reports the percent difference between the solution values of the VND and VNS heuristics.

We can see that the results produced by VNS are always better than those produced by VND, and that half the time the gap between the solution of VNS and the solution of VND is larger than 5% (in bold). In two cases, (brd14051-75 and brd14051-101) this gap exceeds 15%. In general, the improvement produced by VNS decreases on the larger instances where the gap is less than 5%. Regarding CPU time, VND is obviously faster than VNS since its neighborhood is smaller and easier to compute. However, for instances

Instance	Size	RAN_NN	*VND*		*VNS*		Gap (%)
			cost	time	cost	time	
fnl4461	25	5273	2168	0.00	2168.0	0.00	0.00
	51	11634	4301	0.00	4020.0	0.06	**6.53**
	75	15503	6226	0.01	5865.0	0.17	**5.80**
	101	21897	10171	0.04	8852.8	0.70	**12.96**
	251	74129	30927	3.75	29330.6	23.92	**5.16**
	501	275688	77315	86.69	71208.5	458.56	**7.90**
	751	517555	122848	573.37	118383.1	2172.49	3.63
brd14051	25	11758	4779	0.00	4682.2	0.00	2.03
	51	17365	8091	0.00	7763.2	0.06	4.05
	75	28863	8762	0.01	7309.1	0.25	**16.58**
	101	34617	12900	0.05	10005.2	0.74	**22.44**
	251	93483	25469	4.37	24119.3	36.68	**5.30**
	501	219684	57504	82.84	52806.8	478.69	**8.17**
	751	351172	88667	511.24	86230.1	2169.77	2.75
d15112	25	138643	100230	0.00	93981.0	0.00	**6.23**
	51	259473	155554	0.00	143575.2	0.04	**7.70**
	75	445627	221948	0.01	201385.4	0.18	**9.26**
	101	579279	293492	0.04	276876.8	0.46	**5.66**
	251	1354723	621836	3.20	589066.9	24.64	**5.27**
	501	2401410	974488	81.63	953764.5	385.88	2.13
	751	3725560	1425598	444.69	1352866.6	1968.56	**5.10**
d18512	25	11758	4779	0.00	4683.4	0.00	2.00
	51	26039	7894	0.00	7565.6	0.06	4.16
	75	30249	9997	0.01	8781.5	0.20	**12.16**
	101	45269	11314	0.06	10332.4	0.52	**8.68**
	251	106116	28244	4.43	24855.4	32.89	**12.00**
	501	216088	54336	84.96	52295.6	485.96	3.76
	751	340729	86957	569.27	83763.3	2508.53	3.67
nrw1379	25	4368	3356	0.00	3194.8	0.00	4.80
	51	13135	5195	0.00	5095.0	0.05	1.92
	75	17889	7385	0.01	6865.1	0.18	**7.04**
	101	27311	10802	0.04	10197.5	0.53	**5.60**
	251	71614	29178	3.08	27936.2	24.54	4.26
	501	168928	63038	78.56	60584.5	380.11	3.89
	751	281104	110650	462.62	105136.1	2447.09	4.98
pr1002	25	30758	16221	0.00	16221.0	0.00	0.00
	51	67278	36394	0.00	31186.6	0.07	**14.31**
	75	91232	47287	0.01	46911.0	0.28	0.80
	101	138225	65110	0.04	63611.1	0.83	2.30
	251	569919	210595	3.37	200028.5	31.28	**5.02**
	501	1275796	501520	82.65	485042.3	471.86	3.29
	751	2308307	843629	486.26	819197.7	2785.40	2.90

Table 2: Performance comparison between VND and VNS.

with up to 100 vertices the CPU time difference between the two heuristics is negligible since both require less than one second. For VNS this time increases on average to 30 seconds for instances of 251 vertices, to eight minutes for instances of 501 vertices, and to forty minutes for instances of 751 vertices.

Another interesting aspect that we have studied is the impact of each operator on the VNS performance. To this end, each operator was in turn removed from VNS, and all instances were solved by means of the new modified heuristic. These new heuristics are called: *no-CE*, *no-BE*, *no-RB*, *no-MR* and *no-2optL*. Table 3 reports the percent difference between the solution values of the VNS and those of the modified versions. If this difference

Instance	Size	no-CE	no-BE	no-ℜB	no-MR	no-2optL
fnl4461	25	0.00	0.00	0.00	0.00	0.00
	51	3.27	0.83	0.82	1.52	0.00
	75	0.70	0.53	**4.03**	2.18	0.04
	101	-0.66	1.86	3.43	2.93	-0.93
	251	1.14	2.65	2.29	1.78	-0.13
	501	3.71	**6.40**	2.85	3.81	0.88
	751	1.42	**4.11**	2.23	3.15	0.19
brd14051	25	-0.22	-0.07	-0.13	0.75	0.07
	51	0.75	0.39	1.53	2.15	0.36
	75	0.98	-0.44	-0.05	2.42	1.38
	101	1.73	0.18	1.77	3.08	1.55
	251	-0.24	3.74	1.04	3.98	0.44
	501	0.18	3.39	1.85	**4.00**	0.14
	751	0.09	3.14	0.52	2.32	-0.37
d15112	25	0.00	0.10	0.00	2.84	0.39
	51	-0.79	0.12	0.59	1.52	-0.17
	75	1.15	2.55	1.89	**4.70**	0.51
	101	-0.34	-0.62	-0.61	1.74	0.93
	251	-0.02	0.59	1.27	2.77	-0.43
	501	1.15	**4.00**	1.79	2.88	0.39
	751	0.88	**4.07**	2.14	1.89	0.44
d18512	25	-0.24	0.07	-0.03	0.83	-0.01
	51	0.29	0.42	0.05	2.78	-0.17
	75	-0.74	0.40	1.00	2.67	-0.56
	101	1.61	2.19	**5.46**	**5.03**	-0.26
	251	0.97	3.41	2.72	3.08	-0.41
	501	-1.36	3.45	0.61	3.40	0.08
	751	1.16	**4.59**	1.02	3.04	0.52
nrw1379	25	0.04	-0.04	0.18	2.10	-0.09
	51	-0.53	-0.79	-0.79	**4.61**	-0.49
	75	2.54	3.11	2.86	**5.10**	0.07
	101	-0.93	-0.86	-0.12	2.94	0.29
	251	-0.25	1.14	0.21	2.84	-0.17
	501	0.37	2.79	1.23	2.74	0.18
	751	0.22	**4.87**	2.36	2.64	0.34
pr1002	25	0.00	0.00	0.00	0.00	0.00
	51	-0.25	-0.12	**5.56**	0.62	-0.24
	75	0.52	1.61	2.15	**4.71**	0.33
	101	1.58	0.55	3.28	2.06	-0.68
	251	0.58	1.94	2.01	2.06	0.02
	501	0.57	2.53	1.46	2.18	0.39
	751	0.45	3.37	1.36	2.05	-0.27
# Impr.		**13/42**	**7/42**	**6/42**	**0/42**	**16/42**
Average		**0.51**	**1.66**	**1.48**	**2.64**	**0.11**

Table 3: The operators effect on the VNS heuristic.

is negative, then the modified version has produced a better solution than VNS for that instance. Clearly a small number of negative values in each column reflects a greater impact of the operator associated to the column. The second to last line of Table 3 reports the proportion of the times the modified heuristic has produced a better solution than VNS. We can see that the modified heuristics perform worse than the original VNS algorithm. The last line of the table shows the percent average deterioration in the solution value resulting from the removal of each operator. The last two lines of the table clearly justify the use of each operator.

Regarding the impact of each operator, we can see that *multi-relocate* is really crucial to our heuristic because *no-MR* never produces a better solution than VNS. In some cases, the gap between *no-MR* and VNS can be larger than 4% (in bold). The *couple-exchange* and *2-opt-L* operators seem the least useful because *no-CE* and *no-2optL* find better solutions than the base algorithm in 13 and 16 cases, respectively, but the improvement produced is always less than 1%. Finally, *no-BE* and *no-RB* improve the solution 7 and 6 times, respectively, but this improvement is always less than 1%. In some cases, the deterioration resulting from the use of these heuristics is larger than 4%.

6. Conclusion

We have studied a variation of the classical TSP with pickup and delivery in which a LIFO constrained is imposed. This condition allows the elimination of costly reshuffling operations often encountered in practice. This is a relatively new problem on which little research has been carried out. We have introduced new operators whose complexity never exceeds $O(n^3)$. These have been embedded within a variable neighborhood search heuristic. Test results show that our heuristic outperforms previous heuristics.

Acknowledgments

This work was partially supported by the Canadian Natural Science and Engineering Research Council under grants 227837-04 and OGP0039682. We thank Giovanni Righini for providing his VND codes.

References

Carrabs, F. 2005. Heuristics and exact approaches for transportation problems with pickup and delivery. Ph.D. Thesis, Dipartimento di Matematica ed Informatica, Universitá di Salerno, Fisciano, Italy.

Cassani, L., G. Righini. 2004. Heuristic algorithms for the tsp with rear-loading. 35th Annual Conference of the Italian Operational Research Society (AIRO XXXV), Lecce, Italy, September 2004. http://optlab.dti.unimi.it/Papers/Cassani.pdf.

Cordeau, J.-F., G. Laporte, J.-Y.Potvin, M.W.P. Savelsbergh. 2006. Transportation on demand. In C. Barnhart and G. Laporte, eds., *Transportation*. Elsevier, Amsterdam.

Fischetti, M., P. Toth. 1989. An additive bounding procedure for combinatorial optimization problems. *Operations Research* **37** 319-328.

Healy, P., R. Moll. 1995. A new extension of local search applied to the dial-a-ride problem. *European Journal of Operational Research* **83** 83-104.

Kalantari, B., A.V. Hill , S.R. Arora. 1985. An algorithm for the traveling salesman problem with pickup and delivery customers. *European Journal of Operational Research* **22** 377-386.

Ladany, S.P., A. Mehrez. 1984. Optimal routing of a single vehicle with loading and unloading constraints. *Trasportation Planning and Technology* **8** 301-306.

Levitin, G. 1986. Organization of computations that enable one to use stack memory optimally. *Soviet Journal of Computer & System Science* **24** 151-159.

Levitin, G., R. Abezgaouzb. 2003. Optimal routing of multiple-load AGV subject to LIFO loading constraints. *Computers & Operations Research* **30** 397-410.

Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the traveling salesman problem. *Operations Research* **21** 498-516.

Mladenović, N., P. Hansen. 1997. Variable neighborhood search. *Computers & Operations Research* **24** 1097-1100.

Or, I. 1976. Traveling salesman type combinatorial problems and their relations to the logistics of blood banking. Ph.D. Thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, U.S.A.

Pacheco Bonrostro, J.A. 1997. Heuristico para los problemas de ruta con carga y descarga en sistemas lifo. *SORT Statistics and Operations Research Transactions* **21** 69-86.

Psaraftis, H.N. 1983. K-interchange procedures for local search in a precedence-constrained routing problem. *European Journal of Operational Research* **13** 391-402.

Renaud, J., F.F. Boctor, J. Ouenniche. 2000. A heuristic for the pickup and delivery traveling salesman problem. *Computers & Operations Research* **27** 905-916.

Renaud, J., F.F. Boctor, G. Laporte. 2002. Perturbation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research* **29** 1129-1141.

Ruland, K.S., E.Y. Rodin. 1997. The pickup and delivery problem: faces and branch-and-cut algorithm. *Computers and Mathematics with Applications* **33** 1-13.

Savelsbergh, M.W.P. 1990. An efficient implementation of local search algorithms for contrained routing problems. *European Journal of Operational Research* **47** 75-85.

Van Der Bruggen, L.J.J., J.K. Lenstra, P.C. Schuur. 1993. Variable depth search for the single-vehicle pickup and delivery problem with time windows. *Transportation Science* **27** 391-402.

Volchenkov, S.G. 1982. Organization of computations utilizing stack storage. *Engineering Cybernetics, Soviet Journal of Computer & System Science* **20** 109-115.

Xu, H., Z. Chen, S. Rajagopal, S. Arunapuram. 2003. Solving a practical pickup and delivery problem. *Transportation Science* **37** 347-364.